****

**《计算机组成原理》**

**实验报告**

**学生姓名** 余梓俊

**学 号** 2018211991

**专业班级** 计算机科学与技术18-3班

**指导教师** 安鑫

**院系名称** 计算机与信息学院

**2020 年 6 月 19 日**

实验二、CPU部件实现之ALU和寄存器堆

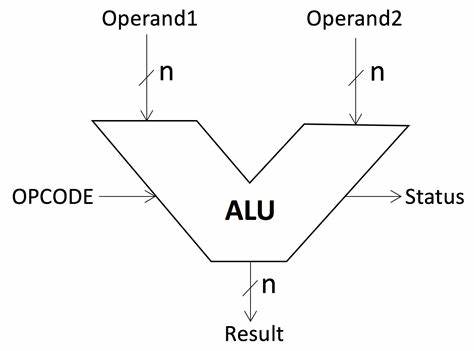
## 实验目的：

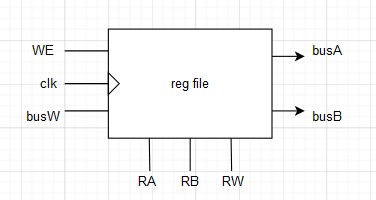
理解和掌握 CPU 中的算术逻辑运算部件（ALU）和寄存器堆（Register File）的工作原理，并使用 Verilog 和 ModelSim 进行设计和仿真。

## 实验内容：

1. 使用 Verilog 完成 ALU 的设计,并编写测试仿真文件验证其正确性。要求：
   * ALU 支持 16 位的加、减、与、或以及移位运算。
2. 使用 Verilog 完成通用寄存器堆的设计，并编写测试仿真文件验证其正确性。要求
   * 寄存器堆包含 8 个 16 位的寄存器；
   * 寄存器堆有两个读端口和一个写端口。

## 实验原理：





ALU原理：

ALU有三个输入，两个数据16位输入（代码中用in1, in2表示），一个3位控制输入（代码中用op表示）。一个16位输出（用out表示）。

寄存器堆原理：

因为有8个寄存器，输入RA，RB，RW为3位。

当时钟上升沿到来时，若写使能信号WE有效，将16位的数据输入busW写入RW信号对应的寄存器。

## 实验步骤：

ALU代码：

module alu(

  input wire [15:0] in1, in2,

  input wire [2:0] op,

  output reg [15:0] out

);

  always@\* begin

    case (op)

      3'b000: out = in1 + in2;

      3'b001: out = in1 - in2;

      3'b010: out = in1 & in2;

      3'b011: out = in1 | in2;

      3'b100: out = in1 << in2;

      3'b101: out = in1 >> in2;

    endcase

  end

endmodule

寄存器堆代码：

module reg\_file(

  input wire WE, clk,

  input wire [2:0] RA, RB, RW,

  input wire [15:0] busW,

  output wire [15:0] busA, busB

);

  reg [15:0] regs[0:7];

  assign busA = regs[RA];

  assign busB = regs[RB];

  always@(posedge clk) begin

    if (WE == 1)

      regs[RW] = busW;

  end

endmodule

ALU测试代码：

In1为：0000111101011010

In2为：0000000001010101

分别测试加减乘除，然后讲In2改为8，分别对In1进行左移右移8位。

module alu\_tb;

  reg [15:0] in1, in2;

  reg [2:0] op;

  wire [15:0] out;

  initial begin

    in1 = 16'b0000111101011010;

    in2 = 16'b0000000001010101;

    op = 3'b000;

    #10

    op = 3'b001;

    #10

    op = 3'b010;

    #10

    op = 3'b011;

    #10

    in2 = 4'h08;

    op = 3'b100;

    #10

    op = 3'b101;

    #10 $stop;

  end

  alu uut(

    .in1(in1), .in2(in2), .op(op),

    .out(out)

  );

endmodule

寄存器堆测试代码：

时钟信号每个5个时间单位取反，即周期为10个时间单位。

RA读取第0号寄存器，RB读取1号寄存器。

通过RW和busW将0号寄存器值设为ff00；10个时间单位后将RW改为1，busW改为00ff，期望将1号寄存器设为00ff，但此时先让WE为0，以测试写使能信号是否正常工作；再过10个时间单位后将WE设为1，此时1号寄存器内容才应变为00ff。

module reg\_file\_tb;

  reg WE, clk;

  reg [2:0] RA, RB, RW;

  reg [15:0] busW;

  wire [15:0] busA, busB;

  always#5 clk = ~clk;

  initial begin

    clk =0;

    RA = 3'b000;

    RB = 3'b001;

    RW = 3'b000;

    busW = 16'hff00;

    WE = 1;

    #10

    RW = 3'b001;

    busW = 16'h00ff;

    WE = 0;

    #10

    WE = 1;

    #10

    $stop;

  end

  reg\_file uut(

    .WE(WE), .clk(clk),

    .RA(RA), .RB(RB), .RW(RW),

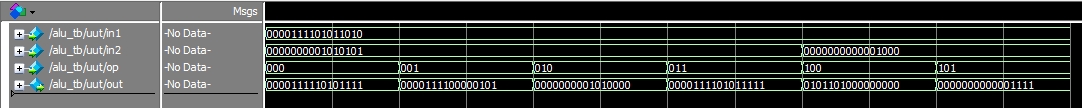
    .busW(busW),

    .busA(busA), .busB(busB)

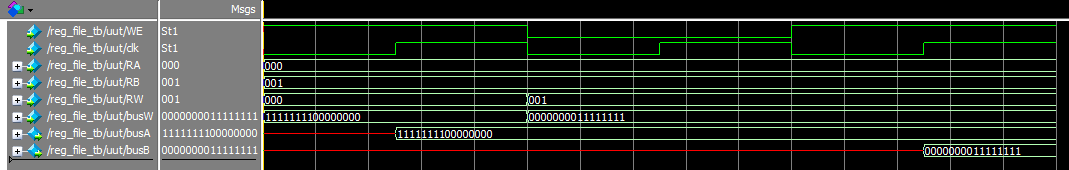
);

endmodule

ALU测试仿真：



寄存器堆测试仿真：



## 总结：

本次实验实践了算术逻辑单元和寄存器堆的软件模拟，加深了对这两个CPU的基本部件的理解。

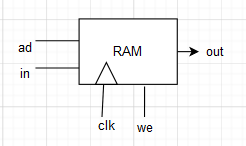
**实验三、CPU 部件实现之 PC 和半导体存储器 RAM**

1. **实验目的：**

理解和掌握 CPU 中程序计数器 PC 和半导体存储器 RAM 的工作原理，并使用 Verilog和ModelSim进行设计和仿真。

1. **实验内容：**
2. 使用 Verilog 完成程序计数器 PC 的设计，要求：
   * PC 为 8 位计数器
3. 使用 Verilog 完成数据存储器的设计，并编写测试仿真文件验证其正确性。要求
   * 存储字长 16 位，存储容量 1K 字节；
   * 一根读写控制信号线控制读写，低电平有效。
4. **实验原理：**





PC，程序计数器，在上升沿到来时，若reset有效，置为0，否则自增1。

RAM，随机存取存储器，代码实现类似寄存器堆。1K字节，字长16位，因此对应9根地址线，字地址从九个0到九个1。

1. **实验步骤：**

PC代码：

module pc(

  input wire clk, reset,

  output reg[7:0] pc

);

  always@(posedge clk) begin

    if (reset == 1) pc = 0;

    else pc = pc + 1;

  end

endmodule

RAM代码：

module ram(

  input wire clk, WE,

  input wire[8:0] ad,

  input wire[15:0] in,

  output wire[15:0] out

);

  reg[15:0] words[511:0];

  assign out = words[ad];

  always@(posedge clk) begin

    if (WE == 0)

      words[ad] = in;

  end

endmodule

PC测试代码：

module pc\_tb;

  reg clk, reset;

  wire[7:0] pc;

  always#5 clk = ~clk;

  initial begin

    clk = 0;

    reset = 1;

    #10

    reset = 0;

    #80

    reset = 1;

    #20

    reset = 0;

    #40

    $stop;

  end

  pc uut(

    .clk(clk),

    .reset(reset),

    .pc(pc)

  );

endmodule

RAM测试代码：

module ram\_tb;

  reg clk, WE;

  reg[8:0] ad;

  reg[15:0] in;

  wire[15:0] out;

  always#5 clk = ~clk;

  initial begin

    clk = 0;

    WE = 0;

    ad = 0;

    in = 16'h000f;

    #10

    ad = 1;

    in = 16'h00f0;

    #10

    WE = 1;

    ad = 0;

    #10

    ad = 1;

    #10

    $stop;

  end

  ram uut(

    .clk(clk), .WE(WE),

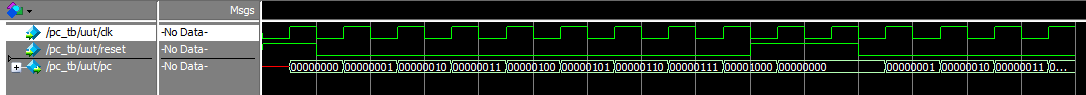
    .ad(ad), .in(in),

    .out(out)

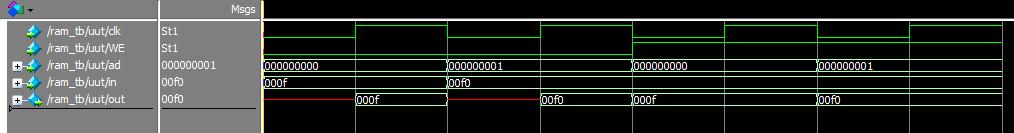
  );

endmodule

PC测试模拟结果：



RAM测试模拟结果：



1. **总结：**

实验三加深了我对程序计数器和随机存取存储器的工作方式的理解。过程中还遇到了Verilog语法上的小坑：16位2进制，即4位16进制，书写时不是4’h00f0,而是16’h00f0，其实该问题已经在实验二时遇到过，但由于实验二的完成时间太早，又忘记了这个问题。

**实验四、单周期CPU设计与实现——单指令CPU**

1. **实验目的：**

通过设计并实现支持一条指令的 CPU，理解和掌握 CPU 设计的基本原理和过程。

1. **实验内容：**

设计和实现一个支持**加法指令**的单周期 CPU。要求该加法指令（表示为**add r1，r2，r3**）格式约定如下：

* 采用寄存器寻址，r1，r2，r3 为寄存器编号，r1 和 r2 存放两个源操作数，r3为目标寄存器，其功能为[r1] + [r2] -> r3；
* 指令字长 16 位，操作码和地址码字段分配如下所示：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 15 | 9 | 8 | 6 | 5 | 3 | 2 | 0 |
| OpCode |  | r1 |  | r2 |  | r3 |  |

1. **实验原理：**

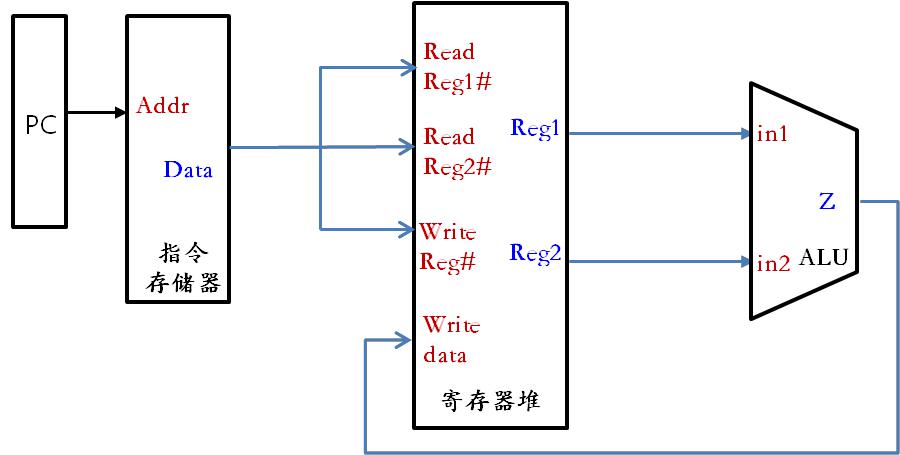
单周期 CPU 是指所有指令均在一个时钟周期内完成的 CPU。CPU 由数据通路及其控制部件两部分构成，因而要完成一个支持若干条指令 CPU 的设计，需要依次完成以下两件事：

1） 根据指令功能和格式设计 CPU 的数据通路；

2） 根据指令功能和数据通路设计控制部件。

**3.1 根据功能和格式完成 CPU 的数据通路设计**

本实验需要设计的 CPU 只需要支持一条加法指令，而该指令的功能是在一个时钟周期内从寄存器组中 r1 和 r2 中取出两个操作数，然后送到 ALU 进行加法运算，最后把计算结果保存到 r1 寄存器中。下图给出了改加法指令的数据通路图。



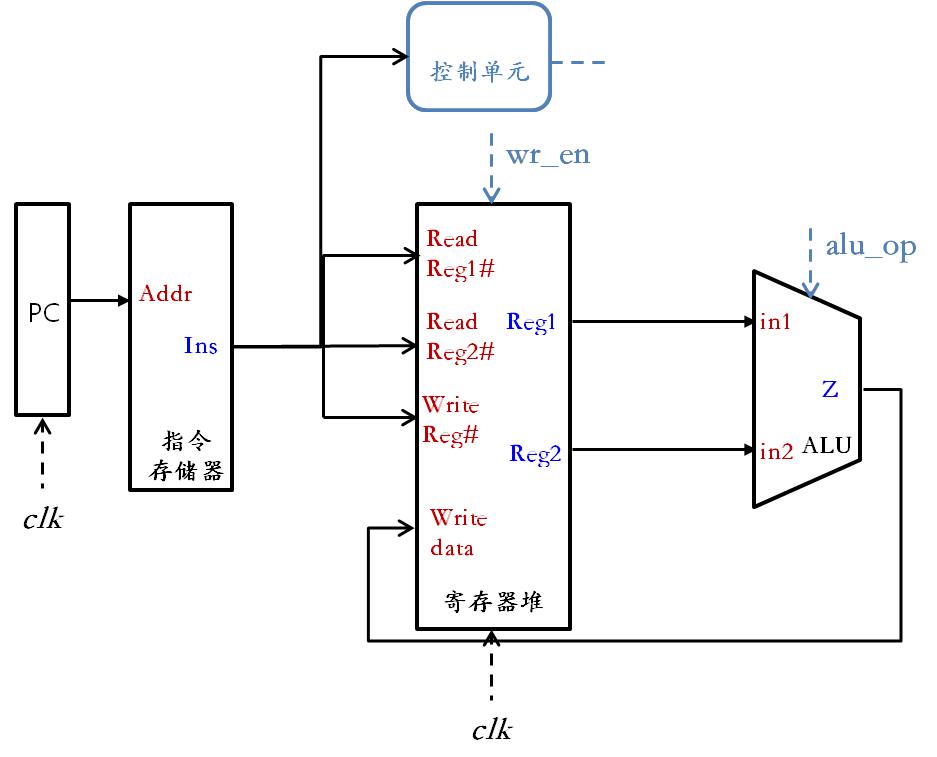
* **3.1. 加法指令 add r1,r2,r3 数据通路**

此外，还需要确定各个部件的位数，为了简单起见，我们假设目标 CPU 的机器字长、存储字长和指令字长相等均为 16 位，存储单元个数假设为 256，按字寻址，并取 PC 位数为8。

**3.2 根据指令功能、数据通路完成控制单元的设计**

控制单元的功能是为当前要执行的指令产生微操作命令从而完成该指令的执行。为了能够完成加法指令的执行，结合**图 1**，控制单元需要在取出指令后根据指令操作码（本例中是加法指令），控制 ALU（参考实验二）做加法（通过给 alu\_op 信号线相应赋值），并把结果写回寄存器组（参考实验三）中（通过给 wr\_en 赋值为 true）。

图 2 给出了整合控制单元后目标 CPU 的原理图，系统时钟信号也已标注。



* **3.2. 单指令 CPU 原理图**

1. **实验步骤：**

在第三部分通过对该 CPU 实现细节的分析、设计，并得到该 CPU 的原理图后，就可以依次实现各个模块，并进行仿真验证了。

**4.1 CPU 各模块 Verilog 实现**

在前面实验中，已经分别设计和实现了 PC、指令存储器、寄存器组和 ALU，这里只给出各个模块的功能描述及其接口定义，具体实现可以直接使用或者调整前面试验的实现代码。

**1） PC 模块**

表 4.1 PC 模块功能描述

**输入** 时钟信号 clk、重置信号 rst

**输出** 指令地址 pc（8 位）

**功能** 每个时钟上升沿 PC 的值自动加 1，并输出

Verilog 关键代码：

module pc(

  input wire clk, rst,

  output reg[7:0] pc

);

  always@(posedge clk) begin

    if (rst == 1) pc = 0;

    else pc = pc + 1;

  end

endmodule

**2） 指令存储器模块**

表 4.2 指令存储器模块功能描述

**输入** 8 位指令地址 Addr

**输出** 16 位指令 Ins

**功能** 存放待执行的指令（初始化），并根据地址输出指令

Verilog 关键代码：

module insMemory(

  input wire[7:0] Addr,

  output reg[15:0] Ins

);

  reg[15:0] instructions[255:0];

  integer i;

  initial begin

    for (i = 0; i < 256; i = i + 1) begin

      instructions[i][2:0] = (i + 2) % 8;

      instructions[i][5:3] = (i + 1) % 8;

      instructions[i][8:6] = i % 8;

      instructions[i][15:9] = 0;

    end

  end

  always@\* begin

    Ins = instructions[Addr];

  end

endmodule

**3） 寄存器堆**

表 4.3 寄存器堆模块功能描述

|  |  |
| --- | --- |
| **输入** | 时钟信号 clk、读写控制线 wr\_en、读寄存器编号 read\_reg1 和 |
|  | read\_reg2、写寄存器编号 write\_reg、写入数据 write\_data |
|  |  |
| **输出** | 对应两个读寄存器编号的寄存器值 reg1 和 reg2 |
|  |  |
| **功能** | 根据读寄存器编号给出对应寄存器的值；在写允许情况下，把写入 |
|  | 端的数据在 clk 下降沿写到写寄存器编号对应的寄存器 |
|  |  |

Verilog 关键代码：

module reg\_file(

  input wire clk, wr\_en,

  input wire[2:0] read\_reg1, read\_reg2, write\_reg,

  input wire[15:0] write\_data,

  output reg[15:0] reg1, reg2

);

  reg[15:0] regs[7:0];

  integer i;

  initial begin

    for (i = 0; i < 8; i = i + 1) begin

      regs[i] = 1;

    end

  end

  always@\* begin

    reg1 <= regs[read\_reg1];

    reg2 <= regs[read\_reg2];

  end

  always@(negedge clk) begin

      if(wr\_en == 1)

        regs[write\_reg] = write\_data;

  end

endmodule

**4） ALU**

* 4.4 ALU 模块功能描述

**输入** 操作数 in1 和 in2、操作选择信号 alu\_op

**输出** ALU 运算结果 Z

**功能** 根据操作选择信号计算 in1 和 in2 的运算结果 Z

Verilog 关键代码：

module alu(

  input wire[15:0] in1, in2,

  input wire[2:0] alu\_op,

  output reg[15:0] Z

);

  always@\* begin

    case(alu\_op)

      3'b000: Z = in1 + in2;

      3'b001: Z = in1 - in2;

      3'b010: Z = in1 & in2;

      3'b011: Z = in1 | in2;

      3'b100: Z = in1 << in2;

      3'b101: Z = in1 >> in2;

    endcase

  end

endmodule

**5） 控制单元**

|  |  |
| --- | --- |
|  | 表 4.5 控制单元模块功能描述 |
|  |  |
| **输入** | 指令（操作码） |
|  |  |
| **输出** | 寄存器堆的读写控制线 wr\_en、ALU 的操作选择信号 alu\_op |
|  |  |
| **功能** | 根据当前指令功能对 wr\_en 和 alu\_op 赋值 |
|  |  |

Verilog 关键代码：

module cu(

  input wire[6:0] ins\_op,

  output reg wr\_en,

  output reg[2:0] alu\_op

);

  always@\* begin

    if (ins\_op == 0) begin

      wr\_en = 1;

      alu\_op = 0;

    end

  end

endmodule

**4.2 CPU 顶层文件封装实现**

通过根据图 2 将以上定义的模块进行连接、封装就得到了目标 CPU，该 CPU 的输入为系统时钟信号 clk 和重置信号 reset。

Verilog 关键代码：

module cpu(

  input wire clk, reset

);

  wire wr;

  wire[7:0] insAd;

  wire[15:0] ins;

  wire[2:0] alu\_op;

  wire[15:0] alu\_in1, alu\_in2, alu\_z;

  pc pc(

    .clk(clk), .rst(reset), .pc(insAd)

  );

  insMemory insMemory(

    .Addr(insAd), .Ins(ins)

  );

  cu cu(

    .ins\_op(ins[15:9]), .alu\_op(alu\_op), .wr\_en(wr)

  );

  reg\_file reg\_file(

    .clk(clk), .wr\_en(wr),

    .read\_reg1(ins[8:6]), .read\_reg2(ins[5:3]),

    .reg1(alu\_in1), .reg2(alu\_in2),

    .write\_reg(ins[2:0]), .write\_data(alu\_z)

  );

  alu alu(

    .alu\_op(alu\_op),

    .in1(alu\_in1), .in2(alu\_in2), .Z(alu\_z)

  );

endmodule

**4.3 CPU 模拟仿真**

为了仿真验证所实现的 CPU，需要定义测试文件并在测试文件中对指令存储器和寄存器堆中的相应寄存器的值进行初始化，并通过仿真波形图查看是否指令得到了正确执行。

1）TestBench 关键代码

module cpu\_tb;

  reg clk, reset;

  always#5 clk = ~clk;

  initial begin

    clk = 1;

    reset = 1;

    #10 reset = 0;

    #100 $stop;

  end

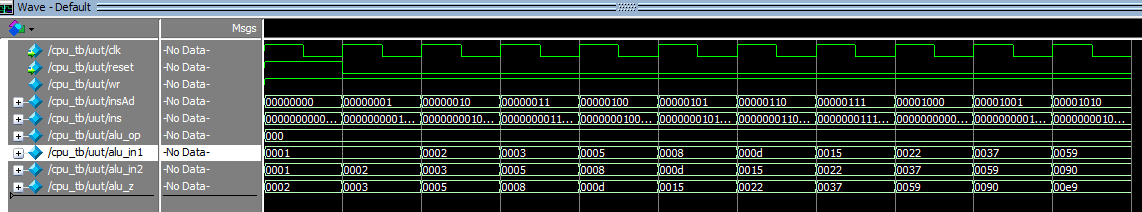
  cpu uut(

    .clk(clk), .reset(reset)

  );

endmodule

2、ModelSim 仿真及分析



1. **总结：**

在指令存储器的初始化部分，由于寄存器的个数为8个，我令第i条指令的两个加数寄存器为i%8和(i+1)%8，结果寄存器为(i+2)%8；而在寄存器堆的初始化过程，我令每个寄存器的初始值均为1。

当指令一条一条的执行下来后，加数在8个寄存器之间不断循环重复，结果的序列（alu\_z）应当为斐波那契数列（从2开始）。从仿真结果中可以看到，结果正确。

**实验五、单周期 CPU 设计与实现——十条指令**

1. **实验目的：**

通过设计并实现支持 10 条指令的 CPU，进一步理解和掌握 CPU 设计的基本原理和过程。

1. **实验内容：**

设计和实现一个支持如下十条指令的单周期 CPU。

* + **非访存指令**
* 清除累加器指令 CLA
* 累加器取反指令 COM
* 算术右移一位指令 SHR：将累加器 ACC 中的数右移一位，结果放回 ACC
* 循环左移一位指令 CSL：对累加器中的数据进行操作
* 停机指令 STP
  + **访存指令**
* 加法指令 ADD X：[X] + [ACC] –〉ACC，X 为存储器地址，直接寻址
* 存数指令 STA X，采用直接寻址方式
* 取数指令 LDA X，采用直接寻址
  + **转移类指令**
* 无条件转移指令 JMP imm：signExt(imm) -> PC

有条件转移（负则转）指令 BAN X: ACC 最高位为 1 则（PC）+ X -> PC,否则PC不变

1. **实验原理：**

**分析：**

1. 机器字长=指令字长=存储字长=16
2. 指令的操作码op按上述顺序按2进制依次递增排列，10条指令，共需要4个2进制位，那么地址码/操作数字段为12个2进制位。
3. 方便起见，初始化insMemory（指令存储器）时非访存指令的地址码/操作数字段为全0。
4. 同样为方便起见，由于这10条指令只由一个操作码字段和一个地址码/操作数字段组成，因此将dataMemory（数据存储器）的地址线长度设为12条。
5. 类似的，跳转指令的操作码为12位，方便起见，将insMemory的大小也设为2的12次方。
6. 各模块的输入输出（部分）
7. pc

输入inData：由于有跳转指令，pc模块应该有inData输入，其值直接等于指令的地址码/操作数字段。

输入unConditional：针对无条件跳转的写信号，由cu产生。

输入conditional：针对条件跳转的写信号，由alu产生（cu解析到无条件跳转指令时发出unConditional，解析到条件跳转指令时由ALU进行运算后发出conditional）。

输入stop：，停机信号，由cu产生。

**由于这10条指令不存在哪一条是对某一模块写两次或以上的，而我们要设计的是单周期CPU，因次我们把PC自增安排在时钟上升沿，在上升沿发生后的高电平期间进行读取和运算，然后在接下来的下降沿进行所有写入操作。**

这样对PC的写入也在下降沿，而在上升沿到来时又会自增，因此跳转指令改变的PC应当再减一。

并且对于条件跳转的目标地址的运算，交由pc自己完成。因为alu的两个输入分别来自acc和dataMemory，若想要由alu来计算目标地址，则需要先将形式地址写入acc，计算后再将alu的输出Z写入到PC，而两次写入则意味着至少需要两个周期，不符合单周期CPU的要求。

1. cu

输出wrxxx：对各个模块的写信号，如wrPc，wrAcc。

输出op：操作码输出至alu。

1. alu

输入in1，in2：如前所述，分别来自acc和dataMemory。

输出Z：运算结果，可能送往dataMemory或acc，由cu决定。

输出conditional：即前文提到的条件跳转指令的写信号。

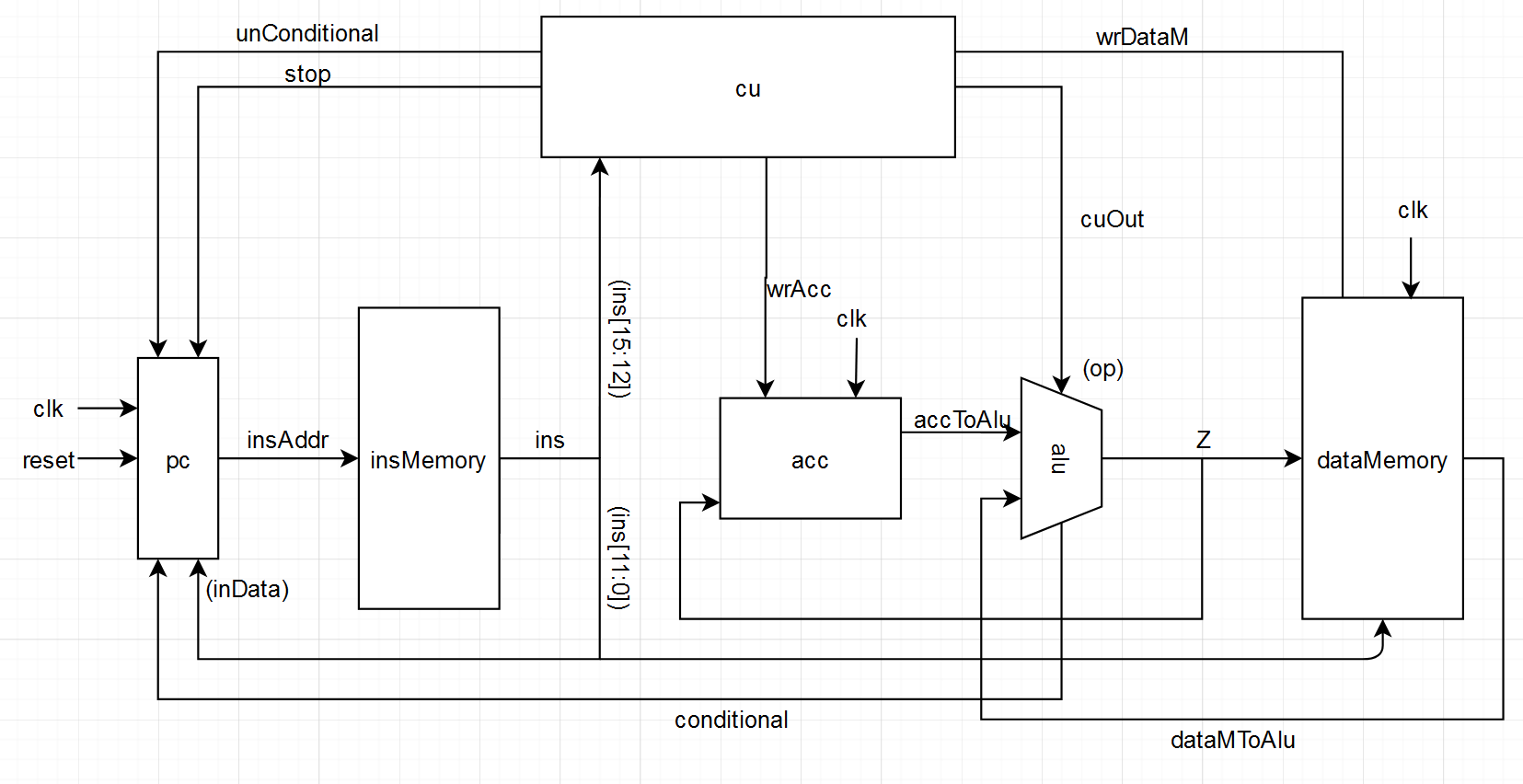
1. acc

alu的结果Z可能会送到acc，存数指令也涉及对acc的数据输入，以及取数指令涉及到对acc的读出和对dataMemory的写入，为了数据通路的简洁性和一致性，我们令dataMemory的输入输出只与alu有关，acc的输入输出也只与alu有关，存数取数指令都经由alu完成。

1. dataMemory

数据存储器dataMomory的数据输入输出已经分析过，对于地址输入，由于这10条指令中不涉及立即数寻址，只有直接寻址，因此地址输入只来自于指令寄存器insMemory。

1. 由于这里的add指令是对acc和直接寻址的存储器数据的操作，不涉及到寄存器堆，因此实验五无需寄存器堆模块。
2. 其余模块以及剩余的输入输出比较简单，且和实验四类似，不再赘述。总的简易图示如下。



1. **实验步骤：**

**cu.v:**

module cu(

  input wire[3:0] opIn,

  output reg[3:0] opOut,

  output reg unConditional, stop,

             wrAcc, wrDataM

);

  always@\* begin

    case (opIn)

      // 0.CLA

      4'b0000: {wrAcc, wrDataM} = 2'b10;

      // 1.COM

      4'b0001: {wrAcc, wrDataM} = 2'b10;

      // 2.SHR

      4'b0010: {wrAcc, wrDataM} = 2'b10;

      // 3.CSL

      4'b0011: {wrAcc, wrDataM} = 2'b10;

      // 4.STOP 4'b0100

      // 5.ADD

      4'b0101: {wrAcc, wrDataM} = 2'b10;

      // 6.STA

      4'b0110: {wrAcc, wrDataM} = 2'b01;

      // 7.LDA

      4'b0111: {wrAcc, wrDataM} = 2'b10;

      // 8.JMP 4'b1000

      // 9.BAN 4'b1001

      default: {wrAcc, wrDataM} = 4'b0000;

    endcase

    opOut = opIn;

    if (opIn == 4'b0100) stop = 1;

    else stop = 0;

    if (opIn == 4'b1000) unConditional = 1;

    else unConditional = 0;

  end

endmodule

**alu.v:**

module alu(

  input wire[3:0] op,

  input wire[15:0] inAcc, inDataM,

  output reg[15:0] Z,

  output reg conditional

);

  always@\* begin

    case (op)

      // 0.CLA

      4'b0000: Z = 0;

      // 1.COM

      4'b0001: Z = ~inAcc;

      // 2.SHR

      4'b0010: begin

        Z[15] <= inAcc[15] == 1 ? 1 : 0;

        Z[14:0] <= inAcc[15:1]; end

      // 3.CSL

      4'b0011: Z = {inAcc[14:0], inAcc[15]};

      // 4.STOP 4'b0100

      // 5.ADD

      4'b0101: Z = inAcc + inDataM;

      // 6.STA

      4'b0110: Z = inAcc;

      // 7.LDA

      4'b0111: Z = inDataM;

      // 8.JMP 4'b1000

      // 9.BAN

      4'b1001: conditional = inAcc[15] == 1 ? 1 : 0;

      default: conditional = 0;

    endcase

  end

endmodule

**acc.v:**

module acc(

  input wire clk, wr,

  input wire[15:0] inData,

  output reg[15:0] acc

);

  always@(negedge clk) begin

    if (wr == 1)

      acc = inData;

  end

  initial begin

    acc = 0;

  end

endmodule

**pc.v:**

module pc(

  input wire clk, reset, stop,

             conditional, unConditional,

  input wire[11:0] inData,

  output reg[11:0] insAd

);

  reg stopped;

  always@\* begin

    if (stop == 1)

      stopped = 1;

  end

  always@(posedge clk) begin

    if (stopped == 1) ;

    else if (reset == 1)

      insAd = 0;

    else insAd = insAd + 1;

  end

  always@(negedge clk) begin

    if (conditional == 1)

      insAd = insAd + inData - 1;

    if (unConditional == 1)

      insAd = inData - 1;

  end

  initial begin

    insAd = 0;

    stopped = 0;

  end

endmodule

**dataMemory.v:**

module dataMemory(

  input wire clk, wr,

  input wire[15:0] inData,

  input wire[11:0] address,

  output reg[15:0] datum

);

  reg[15:0] data[4095:0];

  always@\* begin

    datum = data[address];

  end

  always@(negedge clk) begin

    if (wr == 1)

      data[address] = inData;

  end

  initial begin

    data[0] = 1;

    data[1] = 2;

  end

endmodule

**insMemory.v:**

module insMemory(

  input wire[11:0] insAd,

  output reg[15:0] ins

);

  reg[15:0] instructions[4095:0];

  always@\* begin

    ins = instructions[insAd];

  end

  initial begin

    // 测试过程

    // dataMemory 0号1号内存单元初始值分别为1 2

    // 0.LDA 0    acc = 1

    // 1.CSL      acc = 2

    // 2.BAN 11   不会跳转

    // 3.COM      acc = -3

    // 4.SHR      acc = -2

    // 5.BAN 2    跳转到 7

    // 7.JMP      跳转到 9

    // 9.ADD 1    acc = -2 + 2 = 0

    // 10.STA 0   0号内存单元由1变0

    // 11.LDA 1   acc = 2

    // 12.CLA     acc = 0

    // 13.STP     停机

    // 14.STA 1   由于已停机，1号内存单元不会变为0

    instructions[0] = 16'b0111\_0000\_0000\_0000;

    instructions[1] = 16'b0011\_0000\_0000\_0000;

    instructions[2] = 16'b1001\_0000\_0000\_1010;

    instructions[3] = 16'b0001\_0000\_0000\_0000;

    instructions[4] = 16'b0010\_0000\_0000\_0000;

    instructions[5] = 16'b1001\_0000\_0000\_0010;

    instructions[7] = 16'b1000\_0000\_0000\_1001;

    instructions[9] = 16'b0101\_0000\_0000\_0001;

    instructions[10] = 16'b0110\_0000\_0000\_0000;

    instructions[11] = 16'b0111\_0000\_0000\_0001;

    instructions[12] = 16'b0000\_0000\_0000\_0000;

    instructions[13] = 16'b0100\_0000\_0000\_0000;

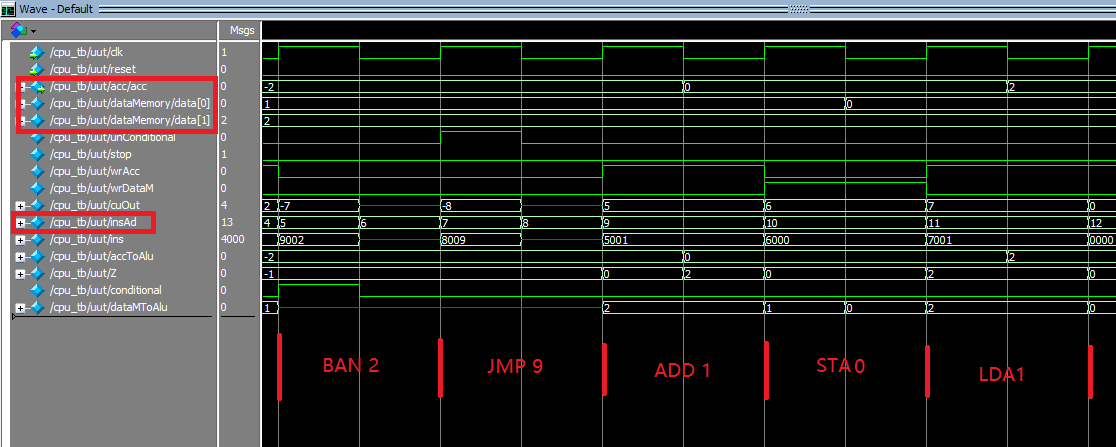
    instructions[14] = 16'b0110\_0000\_0000\_0001;

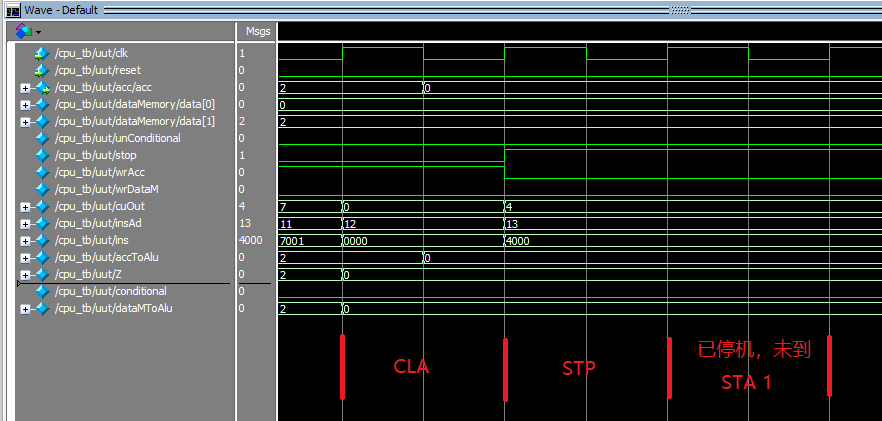
  end

endmodule

**仿真结果：**

****

****

****

观察各个模块间的输入输出数据，均与预期相同。acc的值data[0]、data[1]的值也均与注释中描述的预期相同。

注意到结果图2中，两个跳转指令，中间有一段ins值无效的段，其原因正如前面实验原理中的分析的那样——“我们要设计的是单周期CPU，因次我们把PC自增安排在时钟上升沿，在上升沿发生后的高电平期间进行读取和运算，然后在接下来的下降沿进行所有写入操作”。跳转指令的后半段，pc的值并不是目的地址，而是其减一，而后在下一条指令的开始（上升沿），再增一至目的地址。同时注意到，在这后半段，CU以及ALU相关的值也在发生变化，也就是说CPU执行了我们不需要它执行的指令的分析和运算，但因为紧接着的是上升沿，所以这些错误的计算并不会写入到任何模块。所以，CPU整体的工作的仍然是正确的，只不过在跳转指令的后半段，CPU可能会做多余的无用功。

1. **总结：**

实验五在实验四的基础上迈进了一大步，虽然说整体的模块化思想相同，但由于指令的增加，模块间的通讯复杂度大大地提升了。最初设计时由于对整体工作原理的认识不够深刻，产生了许多错误的设计，在学习了网上的代码后，最终较好地解决了之前的各个问题，一些关键思想已经在原理部分叙述过，不再重复。

实验虽然只用到了Verilog语言一个非常小的子集，但还是踩了一些坑。例如alu中的算数右移代码

       4'b0010: begin

        Z[15] <= inAcc[15] == 1 ? 1 : 0;

        Z[14:0] <= inAcc[15:1]; end

起初忘记应当使用非阻塞赋值“<=”，而是直接使用阻塞赋值“=”。导致仿真时一直存在错误，一度以为是模块的通讯存在漏洞。

总之，五个实验层层递进，完成了这些实验，使我对单周期CPU产生了更深的认识，也体会到了硬件编程的“不便”——调试的不便，与“简洁”——模块通讯只需将各个wire按原理图“连接”起来就可以在同步非同步信号下协同工作，相比软件编程OOP的message passing有一种别样的优雅。